List Homework

- List objects (terms)
- Constructors: cons, nil
- map (that is, map \( f [x_1, \ldots, x_n] = [f x_1, \ldots, f x_n] \))
- foldl (that is, foldl \( f [x_1, \ldots, x_n] = f x_1 (f x_2 (f x_3 (\ldots (f x_n i) \ldots)) \))

\[ \lambda \]

Slide 1

So, what can you do with \( \lambda \) calculus?

\( \lambda \) calculus is very expressive, you can encode:

- logic, set theory
- turing machines, functional programs, etc.

Examples:

\[
\begin{align*}
\text{true} & \equiv \lambda x y. x \\
\text{false} & \equiv \lambda x y. y \\
\text{if} & \equiv \lambda z x y. z x y
\end{align*}
\]

Now, not, and, or, etc is easy:

\[
\begin{align*}
\text{not} & \equiv \lambda x. \text{if } x \text{ false true} \\
\text{and} & \equiv \lambda x y. \text{if } x y \text{ false } \\
\text{or} & \equiv \lambda x y. \text{if } x \text{ true y}
\end{align*}
\]

Slide 3

More Examples

Encoding natural numbers (Church Numerals)

\[
\begin{align*}
0 & \equiv \lambda f x. x \\
1 & \equiv \lambda f x. f x \\
2 & \equiv \lambda f x. f (f x) \\
3 & \equiv \lambda f x. f (f (f x)) \\
\ldots
\end{align*}
\]

Numeral \( n \) takes arguments \( f \) and \( x \), applies \( f \) \( n \)-times to \( x \).

\[
\begin{align*}
iszero & \equiv \lambda n. n (\lambda r. \text{false}) \text{true} \\
succ & \equiv \lambda n. \lambda f x. f (n f x) \\
add & \equiv \lambda m n. \lambda f x. m f (n f x)
\end{align*}
\]

Slide 4
Fix Points

\( (\lambda x. f (x \cdot x)) \) (\( \lambda x. f (x \cdot x) \)) \( \rightarrow^\beta \)

\( (\lambda f. f (x \cdot x)) \) (\( \lambda f. f (x \cdot x) \)) \( \rightarrow^\beta \)

\( t \) (\( (\lambda x. f (x \cdot x)) \) (\( \lambda x. f (x \cdot x) \)) \) \( \rightarrow^\beta \)

\( \mu = (\lambda f. f (x \cdot x)) (\lambda f. f (x \cdot x)) \)

\( \mu t \rightarrow^\beta \mu t \rightarrow^\beta \mu t (t (\mu t)) \rightarrow^\beta \mu t (t (t (\mu t))) \rightarrow^\beta \ldots \)

\( (\lambda x. f (x \cdot x)) (\lambda x. f (x \cdot x)) \) is Turing’s fix point operator

Nice, but ...

As a mathematical foundation, \( \lambda \) does not work. It is inconsistent.

- Frege (Predicate Logic, ~1879): allows arbitrary quantification over predicates
- Russel (1901): Paradox \( \{x | x \notin x \} \)
- Whitehead & Russel (Principia Mathematica, 1910-1913): Fix the problem
- Church (1930): \( \lambda \) calculus as logic, true, false, \( \land \), \( \lor \) as \( \lambda \) terms

With

\[ \begin{align*}
[x \cdot P x] &= \lambda x. P x \\
x &\in M \equiv M x
\end{align*} \]

Problem:

you can write

\( R = \lambda x. \text{not} \,(x \cdot x) \)

and get

\( (R \cdot R) = \beta \text{ not} \,(R \cdot R) \)

We have learned so far...

- \( \lambda \) calculus syntax
- free variables, substitution
- \( \beta \) reduction
- \( \alpha \) and \( \eta \) conversion
- \( \beta \) reduction is confluent
- \( \lambda \) calculus is very expressive (turing complete)
- \( \lambda \) calculus is inconsistent

\( \lambda \) calculus is inconsistent

Can find term \( R \) such that \( R \cdot R =_\beta \text{ not} \,(R \cdot R) \)

There are more terms that do not make sense:

1 2, true false, etc.

Solution: rule out ill-formed terms by using types.
(Church 1940)
Introducing types

Idea: assign a type to each “sensible” λ term.

Examples:

- For term t has type α, write t :: α
- If x has type α then λx.x is a function from α to α.
  Write: (λx.x) :: α → α
- For s t to be sensible:
  s must be function
  t must be right type for parameter
  If s :: α → β and t :: α then (s t) :: β

Syntax for λ→

Terms: t ::= v | c | (t t) | (λx. t)
      v, x ∈ V, c ∈ C, V, C sets of names

Types: τ ::= b | ν | τ → τ
       b ∈ {bool, int, ...} base types
       ν ∈ {α, β, ...} type variables

Context Γ:
Γ: function from variable and constant names to types.

Term t has type τ in context Γ:
Γ ⊢ t :: τ
Examples

\( \Gamma \vdash (\lambda x. x) : \alpha \rightarrow \alpha \)

\( [y \leftarrow \text{int}] \vdash y : \text{int} \)

\( [z \leftarrow \text{bool}] \vdash (\lambda y. z) : \text{bool} \)

\( [] \vdash \lambda f x. f x : (\alpha \Rightarrow \beta) \Rightarrow \alpha \Rightarrow \beta \)

A term \( t \) is well typed or type correct
if there are \( \Gamma \) and \( \tau \) such that \( \Gamma \vdash t : \tau \)

Type Checking Rules

Variables:

\( \Gamma \vdash x : \Gamma(x) \)

Application:

\( \Gamma \vdash t_1 : \tau_1 \cap \Gamma \vdash t_2 : \tau_2 \)

\( \Gamma \vdash (t_1 t_2) : \tau_1 \)

Abstraction:

\( \Gamma[x \leftarrow \tau_1] \vdash t : \tau_2 \)

\( \Gamma \vdash (\lambda x. t) : \tau_1 \Rightarrow \tau_2 \)

Example Type Derivation:

\( \Gamma = [f \leftarrow \alpha \Rightarrow \beta, x \leftarrow \alpha] \)

More complex Example

\( \Gamma \vdash f : \alpha \Rightarrow (\alpha \Rightarrow \beta) \cap \Gamma \vdash x : \alpha \)

\( \Gamma \vdash f x : \alpha \Rightarrow \beta \)

\( \Gamma \vdash \lambda f x. f x x : \alpha \Rightarrow (\alpha \Rightarrow \beta) \Rightarrow (\alpha \Rightarrow \beta) \)

\( \Gamma \vdash \lambda x. f x x : \alpha \Rightarrow \beta \)
More general Types

A term can have more than one type.

Example:
\[ \Gamma \vdash \lambda x. x : \text{bool} \Rightarrow \text{bool} \]
\[ \Gamma \vdash \lambda x. x : \alpha \Rightarrow \alpha \]

Some types are more general than others:
\[ \tau \preceq \sigma \] if there is a substitution \( S \) such that \( \tau = S(\sigma) \)

Examples:
\[ \text{int} \Rightarrow \text{bool} \preceq \alpha \Rightarrow \beta \preceq \beta \Rightarrow \alpha \]

Most general Types

Fact: each type correct term has a most general type

Formally:
\[ \Gamma \vdash t : \tau \Rightarrow \exists \sigma. \Gamma \vdash t : \sigma \land (\forall \sigma'. \Gamma \vdash t : \sigma' \Rightarrow \sigma \preceq \sigma') \]

It can be found by executing the typing rules backwards.

→ type checking: checking if \( \Gamma \vdash t : \tau \) for given \( \Gamma \) and \( \tau \)
→ type inference: computing \( \Gamma \) and \( \tau \) such that \( \Gamma \vdash t : \tau \)

Type checking and type inference on \( \lambda^- \) are decidable.

What about \( \beta \) reduction?

Definition of \( \beta \) reduction stays the same.

Fact: Well typed terms stay well typed during \( \beta \) reduction

Formally:
\[ \Gamma \vdash s : \tau \land s \rightarrow_\beta t \Rightarrow \Gamma \vdash t : \tau \]

This property is called subject reduction

What about termination?

\( \beta \) reduction in \( \lambda^+ \) always terminates.

\( \rightarrow \) is decidable

To decide if \( s \rightarrow_\beta t \), reduce \( s \) and \( t \) to normal form (always exists, because \( \rightarrow_\beta \) terminates), and compare result.

\( \rightarrow \) is decidable

This is why Isabelle can automatically reduce each term to \( \beta \eta \) normal form.
What does this mean for Expressiveness?

Not all computable functions can be expressed in $\lambda^\tau!$

How can typed functional languages then be Turing complete?

Fact:
Each computable function can be encoded as closed, type correct $\lambda^\tau$ term using $Y : (\tau \rightarrow \tau) \rightarrow \tau$ with $Y \; t \rightarrow t \; (Y \; t)$ as only constant.

$\Rightarrow Y$ is called fix point operator
$\Rightarrow$ used for recursion
$\Rightarrow$ lose decidability (what does $Y \; (\lambda x.x)$ reduce to?)

Slide 21